Binary Search Trees (BST): Review

- Each node in tree T is an object x
- Contains attributes:

<table>
<thead>
<tr>
<th>Data</th>
<th>Pointers to other nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.key</td>
<td>x.left</td>
</tr>
<tr>
<td>satellite data</td>
<td>x.right</td>
</tr>
<tr>
<td></td>
<td>x.p (parent)</td>
</tr>
</tbody>
</table>

Point to NIL if empty
- Binomial Property: If x is node:
  - y is pointed to on left: y.key <= x.key
  - y is pointed to the right: y.key >= x.key
Binary Tree: Insert

TREE-INSERT(T, z) // assume z.key, z.left, z.right, z.p

1. y = NIL
2. x = T.root
3. while x ≠ NIL
   4. y = x  // Make a copy of parent
   5. if z.key < x.key  // Traverse left branch
      6. x = x.left
   7. else x = x.right  // Traverse right branch
8. z.p = y  // We’ve found an empty node!
9. if y == NIL  // if tree originally empty
10. T.root = z
11. elseif z.key < y.key  // insert as left child of y
12. y.left = z
13. else y.right = z  // insert as right child of y

Binary Tree Insert: Example

Insert Node z with z.key = 13:
Binary Tree: Height

- For n nodes:
  - Height $O(\lg n)$ if randomly built, but can’t guarantee “randomness”.
  - Height $O(n)$ in worst case

Balanced Trees

- Why do we want to balance trees?
- Red-Black Trees are an example of balanced trees
- Other implementations of balanced trees:
  - AVL trees
  - B-trees
  - B+-trees
  - 2-3 trees
  - Scapegoat tree
  - Treap
**Red-Black Tree**

- BST data structure with extra color field for each node, satisfying the red-black properties:
  1. Every node is either red or black.
  2. The root is black.
  3. Every leaf is black.
  4. If a node is red, both children are black.
  5. Every path from node to descendant leaf contain the same number of black nodes.

**Example**

- Attributes of nodes:
  - key
  - left
  - right
  - p (parent)
  - color

- Note the use of the sentinel T.nil
  - Parent of the root is T.nil
  - All leaves are T.nil

- Leave leaves off!
Properties of RB-Trees

- Black-height $bh(x)$ of a node $x$:
  - Number of black nodes on any simple path from, but not including, a node $x$ down to a leaf

- Theorem: A red-black tree with $n$ internal nodes has height at most $2\lg(n+1)$

Lemma

Lemma: A subtree at any node $x$ contains at least $2^{bh(x)} - 1$ internal nodes (nodes with children), where $bh(x)$ is the black-height of node $x$.

Pf: Induct on height $h(x)$ of $x$!
Theorem:

- Theorem: A red-black tree with n internal nodes has height at most $2\log(n+1)$.

- Proof:

Example

- Color this tree
- Insert 8
- Insert 11
- Insert 10

Properties of RB-Trees
1. Every node is either red or black.
2. The root is black.
3. Every leaf is black.
4. If a node is red, both children are black.
5. Every path from node to descendant leaf contain the same number of black nodes.

https://www.cs.usfca.edu/~galles/visualization/RedBlack.html
Rotations

- Why are rotations necessary in red-black trees?
- How are rotations performed?
- What is the running time of rotation?

- Note: We assume that the appropriate child is not NIL.

Left-Rotate

**LEFT-ROTATE** \((T, x)\)

\[
\begin{align*}
y &= x.right \\
x.right &= y.left \\
\text{if } y.left &\neq T.nil \\
y.left.p &= x \\
y.p &= x.p \\
\text{if } x.p == T.nil \\
T.root &= y \\
\text{elseif } x == x.p.left \\
x.p.left &= y \\
\text{else } x.p.right &= y \\
y.left &= x \\
x.p &= y
\end{align*}
\]

// set y
// turn y’s left subtree into x’s right subtree
// link x’s parent to y
// put x on y’s left
Example

- Rotate left about 9

```
\[7\]
\[5\]  \[9\]
\[8\]  \[12\]
\[11\]
```

Inserting into a RB-Tree

- This is regular binary search tree insertion, color node red.
- Which RB-Tree property could have been violated?
- Complexity?

Properties of RB-Trees
1. Every node is either red or black.
2. The root is black.
3. Every leaf is black.
4. If a node is red, both children are black.
5. Every path from node to descendant leaf contain the same number of black nodes.

```plaintext
RB-INSERT(T, z)

y = T.nil
x = T.root
while x != T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.root = z
else if z.key < y.key
    y.left = z
else y.right = z.
z.left = T.nil
z.right = T.nil
z.color = RED
RB-INSERT-FIXUP(T, z)
```
RB-Insert-Fixup(T,z):

RB-INSERT-FIXUP(T, z)
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK  // case 1
            y.color = BLACK    // case 1
            z.p.p.color = RED  // case 1
            z = z.p.p
        else if z == z.p.right
            z = z.p
            LEFT-ROTATE(T, z)  // case 2
            z.p.color = BLACK  // case 2
            z.p.p.color = RED  // case 3
            RIGHT-ROTATE(T, z.p.p)  // case 3
    else (same as then clause with “right” and “left” exchanged)
        T.root.color = BLACK

Case 1: z’s uncle y is also red

Case 1: y is red

If z is a right child

If z is a left child
Cases 2 and/or 3: z’s uncle y is black

**Case 2:** y is black, z is a right child

**Case 3:** y is black, z is a left child

Note: Case 2 may not apply, but if it does case 2 always morphs into case 3

---

**Example**

- Insert 10
Example

- Insert 15